perlintro -- a brief introduction and overview of Perl

NAME
perlintro -- a brief introduction and overview of Perl

DESCRIPTION
This document is intended to give you a quick overview of the Perl programming language, along with pointers to further documentation. It is intended as a "bootstrap" guide for those who are new to the language, and provides just enough information for you to be able to read other peoples' Perl and understand roughly what it's doing, or write your own simple scripts.

This introductory document does not aim to be complete. It does not even aim to be entirely accurate. In some cases perfection has been sacrificed in the goal of getting the general idea across. You are strongly advised to follow this introduction with more information from the full Perl manual, the table of contents to which can be found in the perltoc manpage.

Throughout this document you'll see references to other parts of the Perl documentation. You can read that documentation using the perldoc command or whatever method you're using to read this document.

What is Perl?
Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

Different definitions of Perl are given in the perl manpage, the perfaq1 manpage and no doubt other places. From this we can determine that Perl is different things to different people, but that lots of people think it's at least worth writing about.

Running Perl programs
To run a Perl program from the Unix command line:

    perl programe.pl

Alternatively, put this as the first line of your script:

    #!/usr/bin/env perl

... and run the script as /path/to/script.pl. Of course, it'll need to be executable first, so chmod 755 script.pl (under Unix).

For more information, including instructions for other platforms such as Windows and Mac OS, read the perlrn run manpage.

Basic syntax overview
A Perl script or program consists of one or more statements. These statements are simply written in the script in a
straightforward fashion. There is no need to have a main() function or anything of that kind.

Perl statements end in a semi-colon:

```perl
print "Hello, world";
```

Comments start with a hash symbol and run to the end of the line

```perl
# This is a comment
```

Whitespace is irrelevant:

```perl
print "Hello, world"
```

... except inside quoted strings:

```perl
# this would print with a linebreak in the middle
print "Hello world";
```

Double quotes or single quotes may be used around literal strings:

```perl
print "Hello, world";
print 'Hello, world';
```

However, only double quotes `"interpolate" variables and special characters such as newlines (\n):

```perl
print "Hello, $name\n";     # works fine
print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```perl
print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

```perl
print("Hello, world\n");
print "Hello, world\n";
```

More detailed information about Perl syntax can be found in the perlsyn manpage.

**Perl variable types**

Perl has three main variable types: scalars, arrays, and hashes.

**Scalars**

A scalar represents a single value:

```perl
my $animal = "camel";
my $answer = 42;
```

Scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types.

Scalar values can be used in various ways:

```perl
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

There are a number of `"magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes, and are documented in the perlvar manpage. The only one you need to know about for now is $_ which is the `"default variable". It’s used as the default argument to a number of functions in Perl, and it’s set implicitly by certain looping constructs.

```perl
print;       # prints contents of $_ by default
```

**Arrays**
An array represents a list of values:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

Arrays are zero-indexed. Here's how you get at elements in an array:

```
print $animals[0];  # prints "camel"
print $animals[1];  # prints "llama"
```

The special variable $#array tells you the index of the last element of an array:

```
print $mixed[$#mixed];  # last element, prints 1.23
```

You might be tempted to use $#array + 1 to tell you how many items there are in an array. Don't bother. As it happens, using @array where Perl expects to find a scalar value ('in scalar context') will give you the number of elements in the array:

```
if (@animals < 5) { ... }
```

The elements we're getting from the array start with a $ because we're getting just a single value out of the array -- you ask for a scalar, you get a scalar.

To get multiple values from an array:

```
@animals[0,1];       # gives ("camel", "llama")
@animals[0..2];      # gives ("camel", "llama", "owl")
@animals[1..$#animals];  # gives all except the first element
```

This is called an "array slice".

You can do various useful things to lists:

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
```

There are a couple of special arrays too, such as @ARGV (the command line arguments to your script) and @_ (the arguments passed to a subroutine). These are documented in the perlvar manpage.

**Hashes**

A hash represents a set of key/value pairs:

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

You can use whitespace and the => operator to lay them out more nicely:

```
my %fruit_color = (  
    apple => "red",  
    banana => "yellow",  
);
```

To get at hash elements:

```
$fruit_color{"apple"};        # gives "red"
```

You can get at lists of keys and values with keys() and values().

```
my @fruits = keys %fruit_colors;
my @colors = values %fruit_colors;
```

Hashes have no particular internal order, though you can sort the keys and loop through them.
Just like special scalars and arrays, there are also special hashes. The most well known of these is \%ENV which contains environment variables. Read all about it (and other special variables) in the perlvar manpage.

Scalars, arrays and hashes are documented more fully in the perldata manpage.

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes.

A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes. The following example shows a 2 level hash of hash structure using anonymous hash references.

```perl
my $variables = {
  scalar  =>  {
    description => "single item",
    sigil => '$',
  },
  array   =>  {
    description => "ordered list of items",
    sigil => '@',
  },
  hash    =>  {
    description => "key/value pairs",
    sigil => '%',
  },
};
print " Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";
```

Exhaustive information on the topic of references can be found in the perlreftut manpage, the perllol manpage, the perlref manpage and the perldsc manpage.

**Variable scoping**

Throughout the previous section all the examples have used the syntax:

```perl
my $var = "value";
```

The `my` is actually not required; you could just use:

```perl
$var = "value";
```

However, the above usage will create global variables throughout your program, which is bad programming practice. `my` creates lexically scoped variables instead. The variables are scoped to the block (i.e. a bunch of statements surrounded by curly-braces) in which they are defined.

```perl
my $a = "foo";
if ($some_condition) {
  my $b = "bar";
  print $a;           # prints "foo"
  print $b;           # prints "bar"
}
print $a;               # prints "foo"
print $b;               # prints nothing; $b has fallen out of scope
```

Using `my` in combination with a `use strict` at the top of your Perl scripts means that the interpreter will pick up certain common programming errors. For instance, in the example above, the final `print $b` would cause a compile-time error and prevent you from running the program. Using `strict` is highly recommended.

**Conditional and looping constructs**

Perl has most of the usual conditional and looping constructs except for case/switch (but if you really want it, there is a Switch module in Perl 5.8 and newer, and on CPAN. See the section on modules, below, for more information about modules and CPAN).

The conditions can be any Perl expression. See the list of operators in the next section for information on comparison and boolean logic operators, which are commonly used in conditional statements.

```perl
if
```
There's also a negated version of it:

```perl
unless ( condition ) {
    ...
}
```

This is provided as a more readable version of `if (!condition)`.

Note that the braces are required in Perl, even if you've only got one line in the block. However, there is a clever way of making your one-line conditional blocks more English like:

```perl
# the traditional way
if ($zippy) {
    print "Yow!";
}

# the Perlish post-condition way
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;
```

### while

```perl
while ( condition ) {
    ...
}
```

There's also a negated version, for the same reason we have `unless`:

```perl
until ( condition ) {
    ...
}
```

You can also use `while` in a post-condition:

```perl
print "LA LA LA
" while 1;          # loops forever
```

### for

Exactly like C:

```perl
for ($i=0; $i <= $max; $i++) {
    ...
}
```

The C style for loop is rarely needed in Perl since Perl provides the more friendly list scanning `foreach` loop.

### foreach

```perl
foreach (@array) {
    print "This element is $_\n";
}
```

```perl
# you don't have to use the default $_ either...
foreach my $key (keys %hash) {
    print "The value of $key is $hash{$key}\n";
}
```

For more detail on looping constructs (and some that weren't mentioned in this overview) see the `perlsyn` manpage.
Built-in operators and functions

Perl comes with a wide selection of built-in functions. Some of the ones we’ve already seen include `print`, `sort` and `reverse`. A list of them is given at the start of the `perlfun` manpage and you can easily read about any given function by using `perldoc -f functionname`.

Perl operators are documented in full in the `perlop` manpage, but here are a few of the most common ones:

**Arithmetic**

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

**Numeric comparison**

- `==` equality
- `!=` inequality
- `<` less than
- `>` greater than
- `<=` less than or equal
- `>=` greater than or equal

**String comparison**

- `eq` equality
- `ne` inequality
- `lt` less than
- `gt` greater than
- `le` less than or equal
- `ge` greater than or equal

(Why do we have separate numeric and string comparisons? Because we don't have special variable types, and Perl needs to know whether to sort numerically (where 99 is less than 100) or alphabetically (where 100 comes before 99).

**Boolean logic**

- `&&` and
- `||` or
- `!` not

(and, or and not aren't just in the above table as descriptions of the operators -- they're also supported as operators in their own right. They're more readable than the C-style operators, but have different precedence to `&&` and friends. Check the `perlop` manpage for more detail.)

**Miscellaneous**

- `=` assignment
- `.` string concatenation
- `x` string multiplication
- `..` range operator (creates a list of numbers)

Many operators can be combined with `a =` as follows:

```
$a += 1;       # same as $a = $a + 1
$a -= 1;       # same as $a = $a - 1
$a .= "\n";    # same as $a = $a . "\n";
```

**Files and I/O**

You can open a file for input or output using the `open()` function. It's documented in extravagant detail in the `perlfun` manpage and the `perlopentut` manpage, but in short:

```
open(INFILE, "input.txt") or die "Can't open input.txt: $!";
open(OUTFILE, ">output.txt") or die "Can't open output.txt: $!";
```
You can read from an open filehandle using the <> operator. In scalar context it reads a single line from the filehandle, and in list context it reads the whole file in, assigning each line to an element of the list:

```perl
my $line = <INFILE>;
my @lines = <INFILE>;
```

Reading in the whole file at one time is called slurping. It can be useful but it may be a memory hog. Most text file processing can be done a line at a time with Perl's looping constructs.

The <> operator is most often seen in a `while` loop:

```perl
while (<INFILE>) {    # assigns each line in turn to $_
    print "Just read in this line: \$_";
}
```

We've already seen how to print to standard output using `print()`. However, `print()` can also take an optional first argument specifying which filehandle to print to:

```perl
print STDERR "This is your final warning,\n";
print OUTFILE $record;
print LOGFILE $logmessage;
```

When you're done with your filehandles, you should `close()` them (though to be honest, Perl will clean up after you if you forget):

```perl
close INFILE;
```

### Regular expressions

Perl's regular expression support is both broad and deep, and is the subject of lengthy documentation in the `perlrequick` manpage, the `perlretut` manpage, and elsewhere. However, in short:

#### Simple matching

```perl
if (/foo/)       { ... }  # true if \$_ contains "foo"
if ($a =~ /foo/) { ... }  # true if $a contains "foo"
```

The `//` matching operator is documented in the `perlop` manpage. It operates on \$_ by default, or can be bound to another variable using the `=~` binding operator (also documented in the `perlop` manpage).

#### Simple substitution

```
s/foo/bar/;               # replaces foo with bar in \$_
$s = s/foo/bar/;           # replaces foo with bar in $s
$s = s/foo/bar/g;          # replaces ALL INSTANCES of foo with bar in $s
```

The `s///` substitution operator is documented in the `perlop` manpage.

#### More complex regular expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. These are documented at great length in the `perlre` manpage, but for the meantime, here's a quick cheat sheet:

- `.` a single character
- `\s` a whitespace character (space, tab, newline)
- `\S` a non-whitespace character
- `\d` a digit (0-9)
- `\D` a non-digit
- `\w` a word character (a-z, A-Z, 0-9, _)
- `\W` a non-word character
- `[aeiou]` matches a single character in the given set
- `[^aeiou]` matches a single character outside the given set
- `(foo|bar|baz)` matches any of the alternatives specified

- `^` start of string
- `$` end of string
Quantifiers can be used to specify how many of the previous thing you want to match on, where `thing` means either a literal character, one of the metacharacters listed above, or a group of characters or metacharacters in parentheses.

\[
\begin{align*}
* & \quad \text{zero or more of the previous thing} \\
+ & \quad \text{one or more of the previous thing} \\
? & \quad \text{zero or one of the previous thing} \\
\{3\} & \quad \text{matches exactly 3 of the previous thing} \\
\{3,6\} & \quad \text{matches between 3 and 6 of the previous thing} \\
\{3,\} & \quad \text{matches 3 or more of the previous thing}
\end{align*}
\]

Some brief examples:

\[
\begin{align*}
/^\d+/ & \quad \text{string starts with one or more digits} \\
/^$/ & \quad \text{nothing in the string (start and end are adjacent)} \\
/({[^\@]}){3}/ & \quad \text{a three digits, each followed by a whitespace character (eg "3 4 5")} \\
/(a.+)/ & \quad \text{matches a string in which every odd-numbered letter is a (eg "abacadaf")}
\end{align*}
\]

```perl
# This loop reads from STDIN, and prints non-blank lines:
while (<>) {
    next if /^$/;
    print;
}
```

**Parentheses for capturing**

As well as grouping, parentheses serve a second purpose. They can be used to capture the results of parts of the regexp match for later use. The results end up in $1, $2 and so on.

```perl
# a cheap and nasty way to break an email address up into parts
if ($email =~ /([^@]+)@(.+)/) {
    print "Username is $1\n";
    print "Hostname is $2\n";
}
```

**Other regexp features**

Perl regexps also support backreferences, lookaheads, and all kinds of other complex details. Read all about them in the perlrquick manpage, the perlretut manpage, and the perlre manpage.

**Writing subroutines**

Writing subroutines is easy:

```perl
sub log {
    my $logmessage = shift;
    print LOGFILE $logmessage;
}
```

What's that `shift`? Well, the arguments to a subroutine are available to us as a special array called `@_` (see the perlvar manpage for more on that). The default argument to the `shift` function just happens to be `@_`. So my `$logmessage = shift;` shifts the first item off the list of arguments and assigns it to `$logmessage`.

We can manipulate `@_` in other ways too:

```perl
my ($logmessage, $priority) = @_;       # common
my $logmessage = @_[0];                 # uncommon, and ugly
```

Subroutines can also return values:

```perl
sub square {
    my $num = shift;
    my $result = $num * $num;
    return $result;
}
```
For more information on writing subroutines, see the perls sub manpage.

OO Perl

OO Perl is relatively simple and is implemented using references which know what sort of object they are based on Perl’s concept of packages. However, OO Perl is largely beyond the scope of this document. Read the perlboot manpage, the perlt oot manpage, the perlt ooct manpage and the perl obj manpage.

As a beginning Perl programmer, your most common use of OO Perl will be in using third-party modules, which are documented below.

Using Perl modules

Perl modules provide a range of features to help you avoid reinventing the wheel, and can be downloaded from CPAN (http://www.cpan.org/). A number of popular modules are included with the Perl distribution itself.

Categories of modules range from text manipulation to network protocols to database integration to graphics. A categorized list of modules is also available from CPAN.

To learn how to install modules you download from CPAN, read the perlmodinstall manpage.

To learn how to use a particular module, use perldoc Module::Name. Typically you will want to use Module::Name, which will then give you access to exported functions or an OO interface to the module.

the perlf aq manpage contains questions and answers related to many common tasks, and often provides suggestions for good CPAN modules to use.

the perlmod manpage describes Perl modules in general. the perlmodlib manpage lists the modules which came with your Perl installation.

If you feel the urge to write Perl modules, the perlnewmod manpage will give you good advice.

AUTHOR

Kirrily ‘‘Skud” Robert <skud@cpan.org>

perlintro -- a brief introduction and overview of Perl